## Until now…

- You didn't have to use Ant when you do unit testing with JUnit.

- You didn't have to use JUnit when you used Ant (build-calc.xml).



## Running Unit Tests in a Build Process

- From now, you will use BOTH Ant and JUnit.
  - You will run your Ant build script to
    - Compile your program(s),
      - e.g., `PrimeGenerator.java`
    - Compile your test class(es),
      - e.g., `PrimeGeneratorTest.java`
    - Run your test class(es) with Junit, and
      - e.g., `PrimeGeneratorTest.class`
    - Run your program (optional)
      - e.g., `PrimeGenerator.class`

- Your code base requires, or depends on, JUnit library files (i.e., JUnit JAR files).
  - Your Ant build script needs to reference those JAR files and set their paths into CLASSPATH
    - so that it can compile and run your test classes



## JUnit JAR Files

- JUnit API JAR files
  - `junit-jupitar-api.jar`
  - `junit-jupitar-engine.jar`
  - `junit-jupitar-params.jar`
  - `apiguardian-api.jar`
  - `opentest4j.jar`
  - DO NOT use `junit-vintage-*.jar` (Those are for JUnit version 4.)

- JUnit Platform JAR files
  - `junit-platform-commons.jar`
  - `junit-platform-engine.jar`
  - `junit-platform.launcher.jar`
  - `junit-platform-runner.jar:`
  - `junit-platform-suite-api.jar`

- Your Ant build script needs to reference ALL these JAR files and set their paths in CLASSPATH.



## How to Make JUnit JAR files Available for Your Build Script

- Use Apache Ivy with Ant
  - https://ant.apache.org/ivy/
  - Install it in addition to Ant (c.f. Ivy's reference manual)

- Ivy does "dependency management" for Ant.
  - Ivy allows your Ant build script to download any external library files (JAR files) to the "lib" directory, by default.
    - so your build script can set their paths into CLASSPATH.

  - ivy.xml lists the JAR files you want to use.
  - Your build script states `<ivy:retrieve/>`
    - This task will download the JAR files listed in ivy.xml into the "lib" directory.

- In ivy.xml, you specify each JAR file you want to use with `<dependency>`.
  - <dependency org="…" name="…" rev="…" />
- Search each JAR file you want at: https://mvnrepository.com/

Home » org.junit.jupiter » junit-jupiter-api

**JUnit Jupiter API**

JUnit Jupiter is the API for writing tests using JUnit 5.

| | |
|---|---|
| License | EPL 2.0 |
| Categories | Testing Frameworks |
| Tags | junit testing api |
| Ranking | #43 in MvnRepository (See Top Artifact<br>#3 in Testing Frameworks |
| Used By | 11,340 artifacts |

Central (58) ICM (2)

| Version | Vulnerabilities | R |
|---|---|---|
| 5.9.1 | | Centr |

Home » org.junit.jupiter » junit-jupiter-api » 5.9.1

**JUnit Jupiter API » 5.9.1**

JUnit Jupiter is the API for writing tests using JUr

| | |
|---|---|
| License | EPL 2.0 |
| Categories | Testing Frameworks |
| Tags | junit testing api |
| HomePage | https://junit.org/junit5/ |
| Date | Sep 20, 2022 |
| Files | pom (3 KB) jar (202 KB) View Al |
| Repositories | Central |
| Ranking | #43 in MvnRepository (See Top A<br>#3 in Testing Frameworks |
| Used By | 11,340 artifacts |

Maven Gradle Gradle (Short) Gradle (Kotlin) SBT

```
<!-- https://mvnrepository.com/artifact/org.jun
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.9.1</version>
```

- Your build script:

```
– <ivy:retrieve/>
  <path id="classpath">
    <pathelement location="bin" />
    <pathelement location="test/bin" />
    <fileset dir="lib">
  </path>

  <javac …>
  <classpath refid="classpath"/>
      …
  </javac>

  <junitlauncher …>
  <classpath refid="classpath"/>
      …
  </junitlauncher>
```

  - Useful task (for debugging) to print CLASSPATH
    - `<echo message="${toString:classpath}" />`



- Use the <junitlauncher> task in Ant to run JUnit.
  - Refer to Ant's and JUnit's manuals.
    - https://ant.apache.org/manual/Tasks/junitlauncher.html

```
– <junitlauncher printSummary="true">
      <classpath refid="classpath" />
      <test outputdir="test"
            name="edu.umb.cs680.hw01.CalculatorTest"/>
      <listener type="legacy-plain" sendSysOut="true"/>
  </junitlauncher>

– <junitlauncher printSummary="true">
      <classpath refid="classpath" />
      <testclasses outputdir="test">
          <fileset dir="${test.bin}">
              <include name="edu/umb/cs680/hw01/*Test.class"/>
          </fileset>
          <listener type="legacy-plain" sendSysOut="true"/>
      </testclasses>
  </junitlauncher>
```

  - Use this (the second) option.



# Exercise

- Extend your HW0 solution. Set up the following directory structure
  - <proj dir>
    - hw00.xml
    - ivy.xml
    - src [source code directory]
      - edu/umb/cs680/hw00/Calculator.java
    - bin [binary code directory]
      - Calculator.class will be placed under this directory.
    - test [test code directory]
      - src
        - » edu/umb/cs680/hw00/CalculatorTest.java
      - bin
        - » CalculatorTest.class will be placed under this directory.

- Do: ant -f hw00.xml
  - Will create the "lib" directory.
  - Will download required JAR files into the "lib" directory.
  - Will add the "lib" directory to CLASSPATH
  - Compile source code.
  - Run a test class.



# HW 2

- Re-do HW 1 with Ant, Ivy and JUnit.
  - In HW 1, you didn't use Ant and Ivy.
  - Use Ant and Ivy to
    - Download JUnit JAR files.
    - Do pre-compilation configurations (e.g., directory structures, CLASSPATH settings, etc.)
    - Compile `PrimeGenerator` and `PrimeGeneratorTest`
    - Run test cases (test methods) in `PrimeGeneratorTest`
  - Follow the directory structure you used for HW 1.
- **FIRM** deadline: Nov 6 (Sun), midnight
  - For each HW, do your best to turn in your solution in a week or two.
    - Will give you extra points if you regularly do so.



- Make sure that your build script runs properly.
  - on both your shell and IDE
- Turn in:
  - build script
  - ivy.xml
  - "src" directory
  - "test/src" directory
- DO NOT include binary files (the "lib" directory and .class files).
  - Configure `.gitignore` if you like.
  - I will NOT grade your work if you do that.



# JUnit API (cont'd)

# Assertions.assertEquals()

- `org.junit.jupiter.api.Assertions`
  - Contains a series of *static* assertion methods.
  - `assertEquals( int expected, int actual )`
  - `assertEquals( float expected, float actual )`
    `...`
  - `assertEquals( Object expected, Object actual )`
    - » Defined for each primitive type and `Object`
    - » Returns if two values (expected and actual values) match.
      - »
        ```
        float expected = 12;
        float actual = cut.multiply(3,4);
        assertEquals( expected, actual );
        ```
    - » Throws an `org.opentest4j.AssertionFailedError` if two values do not match.
      - » JUnit catches it; your test cases don't have to.
    - » JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`



# Equality and Identity

- `assertEquals( Object expected, Object actual )`
  - Asserts that `actual` is *logically equal* to `expected`
    - » By calling `expected.equals(actual)`.
      - » c.f. `Object.equals()`
- `assertSame( Object expected, Object actual )`
  - Asserts that `expected` and `actual` refer to the *identical object*
    - » by checking if `expected.hashCode()==actual.hashCode()`
    - »
      ```
      Foo f = new Foo();
      assertSame(f, f);                // PASS
      ```
    - »
      ```
      Singleton instance1 = Singleton.getInstance();
      Singleton instance2 = Singleton.getInstance();
      assertSame(instance1, instance2);  // PASS
      ```



```
String str = "umb";          // Syntactic sugar for
                             //     String str = new String("umb");
                             // str contains a pointer (or reference) to
                             // the String instance.

String expected = str;       // expected and actual refer to the
String actual = str;         // identical String instance.
```

```
assertSame(expected, actual);           // PASS
assertEquals(expected, actual);         // PASS
```

- `assertSame()` checks whether
  - `expected.hashCode()==actual.hashCode()` is true.
- `assertEquals()` checks whether
  - `expected.equals(actual)` returns true.
    - `String.equals()` overrides `Object.equals()` and returns true if two String instances contain the same String value.



```
String expected = "umb";    // Syntax sugar for:
                            //     String expected = new String("umb");
String actual = "umb0".substring(0,2);
                            // Syntax sugar for:
                            //     String temp = new String("umb0");
                            //     temp.substring(0, 2);
                            //     actual = temp;
                            // "umb0" -> "umb"
                            // expected and actual refer to
                            // different String instances.
```

```
assertSame(expected, actual);           // FAIL
assertEquals(expected, actual);         // PASS
```

- `assertSame()` checks whether
  - `expected.hashCode()==actual.hashCode()` is true.
- `assertEquals()` checks whether
  - `expected.equals(actual)` returns true.
    - `String.equals()` overrides `Object.equals()` and returns true if two String instances contain the same String values.

# HW 3

- Write and test a *singleton* class
  - Verify `getInstance()` returns a non-null value
    - Use `Assertions.assertNonNull()`
  - Verify `getInstance()` returns the identical instance when it is called multiple times.
    - Use `hashCode()` and `assertEquals()`
      - Use `assertSame()` alternatively.
- Deadline: Nov 6 (Sun) midnight



# `Object.equals()`

- `Object.equals(Object obj)` compares two objects with:
  - ```
    if( this.toString()==obj.toString() ){ return true; }
    else if{ return false; }
    ```
  - `Object.toString()` returns the identity of an object.
    - String data that consists of an object ID, a class name and a package name.
      - e.g., edu.umb.cs680.junit5intro.Calculator@2b2948e2
  -
  - Performs identity check (not equality check).
    - Even though the method name says "equals."



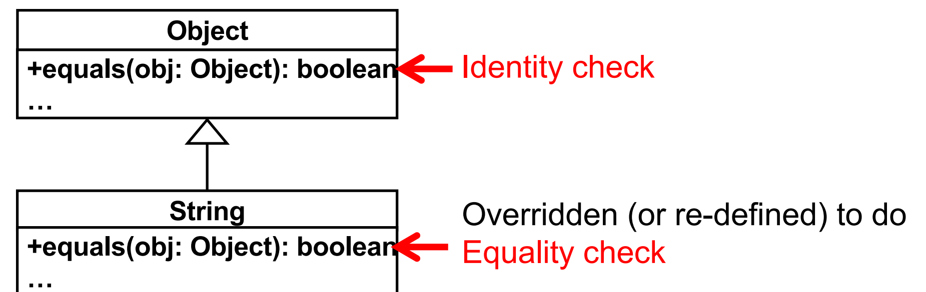
# `Object.equals()`

- `Object.equals(Object obj)` compares two objects with:
  - ```
    if( this.toString()==obj.toString() ){ return true; }
    else if{ return false; }
    ```
  - `Object.toString()` returns the identity of an object.
    - String data that consists of an object ID, a class name and a package name.
      - e.g., edu.umb.cs680.junit5intro.Calculator@2b2948e2
    - Performs identity check (not equality check).
      - Even though the method name says "equals."
- Most Java API classes (e.g. String) override `Object.equals()` to perform appropriate equality check.
  - However, user-defined classes DO NOT… to be discussed.



# `equals()` in Java API

- Most Java API classes override `Object.equals()` to perform appropriate equality check.
  - e.g., `String` overrides `Object.equals()` and returns true if two String instances contain the same String values.




Read the source code of `String.equals()` if you are interested.

# Equality Check for User-defined Classes

- When you define your own class, it is implicitly treated as a subclass of **`Object`**
  - Your class inherits **`Object.equals()`**.
- Your class's **`equals()`** does identity check by default
  - Unless you override **`equals()`**.



```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
```

```
assertSame(p1, p1);       // PASS
assertSame(p1, p2);       // FAIL
assertEquals(p1, p2);     // FAIL
assertEquals(p1, p3);     // FAIL
assertEquals(p2, p3);     // FAIL
```

- **`Person`** inherits **`Object.equals()`**. The inherited method performs *identity check* by default for **`Person`** instances.
  - You need to override **`equals()`** in **`Person`** if you want equality check.

| Person |
|---|
| - firstName: String<br>- lastName: String |
| + Person(first:String, last:String)<br>+ getFirstName(): String<br>+ getLastName(): String |



```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
```

```
assertSame(p1, p2);       // FAIL
assertEquals(p1, p2);     // PASS
assertSame(p1, p3);       // FAIL
assertEquals(p1, p3);     // FAIL
```

| Person |
|---|
| - firstName: String<br>- lastName: String |
| + Person(first:String, last:String)<br>+ getFirstName(): String<br>+ getLastName(): String<br>+ equals(anotherPerson:Object): boolean |

```
if( this.firstName.equals(((Person)anotherPerson).getFirstName())
    && this.lastName.equals(((Person)anotherPerson).getLastName())){
  return true;
}
else{
  return false;
}
```



- Define **`equals()`** in **`Person`**, if your team has a consensus about the equality of **`Persons`**.
- If the consensus may often change, or if there is no reasonable consensus…
  - you should craft equality-check logic in your test class, not in **`Person`**.

```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
assertEquals(p1.getFirstName(), p2.getFirstName());     // PASS
assertEquals(p1.getLastName(), p2.getLastName());       // PASS

assertNotEquals(p1.getFirstName(), p3.getFirstName()); // PASS
assertNotEquals(p1.getLastName(), p3.getLastName());   // PASS
```

- JUnit judges that a test method (test case) passes if it normally returns (i.e., if all four assertion methods return) without **`AssertionFailedError`**

# How to Write Equality-check Logic

- As you use more information for an equality check, you need to call assertion methods more often in a single test method.
  - e.g., first and last names, DOB, zip code for home address.
    - Need to call `assertEquals()` 4 times.
  - e.g., car name, manufacturer name, production year
    - Need to call `assertEquals()` 3 times.
- Equality-check logic gets less clear.
- In general, it makes more sense to perform equality-check by calling assertion methods less often.
  - Consider a String-to-String or array-to-array comparison.



# String-to-String Comparison

```
@Test
... checkPersonEqualityWithJohnJane(){
  Person p1 = new Person("John","Doe",
                         LocalDate...,
                         02125);
  Person p2 = new Person("Jane", "Doe",
                         LocalDate...,
                         02125);

  assertEquals(p1.getFirstName(),
               p2.getFirstName());
  assertEquals(p1.getLastName(),
               p2.getLastName());
  assertEquals(p1.getDOB(),
               p2.getDOB());
  assertEquals(p1.getZipCode(),
               p2.getZipCode());
}
```

```
private String eol =
    System.getProperty("line.separator");

private String personToString(Person p){
  return p.getFirstName() + eol +
         p.getLastName() + eol +
         p.getDOB().toString() + eol +
         p.getZipCode() + eol; }

private String concatenamePersonInfo(
                              String[] p){
  String personInfo;
  for(String info: p){
    personInfo += info + eol;} }

@Test
... checkPersonEqualityWithJohnJane(){
  String[] expectedArray =
      {"John", "Doe", ..., "02125"};
  String expected =
      concatenatePersonInfo(expectedArray);

  Person actual = new Person(
              "John","Doe", ..., 02125);

  assertEquals(expected,
               personToString(actual) ); }
```



# Array-to-Array Comparison

```
private String[] personToStringArray( Person p ){
  String[] personInfo = {p.getFirstName(),
                         p.getLastName(),
                         p.getDOB().toString(),
                         p.getZipCode() };
  return personInfo;
}

@Test
public void checkPersonEqualityWithJohnJane(){
  String[] expected = {"John", "Doe", ..., "0215"};

  Person actual = new Person("John","Doe", ..., 02125);

  assertArrayEquals(expected,
                    personToStringArray(actual) );
}
```



# HW 4

- Define the `Car` class and implement its getter methods.

```
public class Car {
  private String make, model;
  private int mileage, year;
  private float price;  }
```

- Write a test class (`CarTest`) with JUnit
  - Include a private method `carToStringArray()`
  - Define a test method `verifyCarEqualityWithMakeModelYear()`
    - Create two `Car` instances and check their equality with array-to-array comparison
      - Use make, model and year in equality-check logic

```
String[] expected = {"Toyota", "RAV4", "2018"};
Car actual = new Car(...);
assertArrayEquals(expected,
                  carToStringArray(actual) );
```

- Deadline: Nov 6 midnight